BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

THIS PAGE BLANK (USPTO)

89/3 #412

BUNDESREPUBLIK DEUTSCHLAND

PCT/EP 00/09267

EP00/9267

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN
COMPLIANCE WITH
RULE 17.1(a) OR (b)

4

Prioritätsbescheinigung über die Einreichung einer Patentanmeldung

Aktenzeichen: 199 45 940.1

Anmeldetag: 24. September 1999

Anmelder/Inhaber: Infineon Technologies AG, München/DE

Erstanmelder: Siemens AG, München/DE

Bezeichnung: Verfahren und Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur

IPC: G 06 F 9/38

Die angehefteten Stücke sind eine richtige und genaue Wiedergabe der ursprünglichen Unterlagen dieser Patentanmeldung.

München, den 22. Dezember 2000
Deutsches Patent- und Markenamt
Der Präsident
Im Auftrag

Nietl

A 9161
02/00
EDV-L

Beschreibung

Verfahren und Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur

Die vorliegende Erfindung betrifft ein Verfahren und eine Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur.

Die Anzahl der benötigten Zyklen für die Ausführung bestimmter Befehle ist eines der wichtigsten Leistungsmerkmale eines Prozessors. Um maximale Verarbeitungsgeschwindigkeit und minimalen Leistungsverbrauch zu erreichen, soll die Anzahl der Zyklen möglichst minimiert werden. Zu diesem Zweck kennt der Stand der Technik bereits Prozessoren mit der sogenannten "Pipelined"-Architektur. Dies bedeutet, daß der Prozessor mehrere Befehle gleichzeitig abarbeitet, wobei sich jeder Befehl in einer anderen Stufe der Bearbeitung befindet. Beispielsweise wird ein Befehl gerade ausgeführt, der nächste wird gleichzeitig schon decodiert, der übernächste aus dem Speicher angefordert, etc.

In einer solchen "Pipelined"-Architektur kann insbesondere eine bedingter Sprungbefehl (branch) zum sogenannten "harzard" führen, wodurch dann sogar falsche Ergebnisse entstehen können. Bei einem bedingten Sprungbefehl liegt nämlich erst nach Abarbeitung dieses bedingten Sprungbefehls die Adresse des nächstfolgenden Befehls fest. Auf diese Weise kann also der nächstfolgende Befehl erst dann aus dem Speicher angefordert und decodiert werden, wenn das Ergebnis der Ausführung des vorigen Befehls aus dem Rechenwerk des Prozessors zur Verfügung steht.

Gemäß dem bisherigen Stand der Technik wurde dieses "hazard"-Problem dergestalt gelöst, daß direkt nach dem Sprungbefehl so viele Dummy-Befehle (NOP), also No-Operation- oder Wartebefehle eingefügt werden, daß die Ergebnisse auf jeden Fall

richtig bleiben. Dadurch werden allerdings auch so viele Prozessorzyklen nicht ausgenutzt, wie Dummy-Befehle abgearbeitet werden müssen.

Es ist daher die Aufgabe der vorliegenden Erfindung, die Bearbeitung bedingter Sprungbefehle in einem Prozessor mit „pipelined"-Architektur ohne einen so großen Verlust an Prozessorzyklen durch Dummybefehle zu ermöglichen.

Erfindungsgemäß wird diese Aufgabe durch ein Verfahren zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur gelöst, bei der jedem Befehl, nach dem ein bedingter Sprung ausgeführt werden soll, ein oder mehrere zusätzliche Bits hinzugefügt werden, die angeben, unter welcher Bedingung der bedingte Sprung auszuführen ist. Auf dieses Weise kann bereits ein Befehl früher festgestellt werden, ob eine Verzweigung (branch) durchzuführen ist, oder nicht. Damit steht bereits ein Befehl früher fest, welches der nächste Befehl nach dem bedingten Sprung sein wird. Durch diese "branch-prediction" im Befehlssatz ist es also möglich, wesentlich früher das Sprungziel eines bedingten Sprungbefehls festzustellen.

Dabei ist es besondere bevorzugt, daß zusätzlich zu jedem Befehl, nach dem ein bedingter Sprung ausgeführt werden soll, die entsprechende Sprungadresse zugefügt wird. Auf diese Weise ist einen Befehl früher nicht nur bekannt, ob ein bedingter Sprung durchgeführt werden soll oder nicht, sondern es ist bereits die entsprechende neue Zieladresse bekannt. Damit kann bereits der richtige Befehl aus dem Arbeitsspeicher des Prozessors angefordert werden.

Weiter können vorzugsweise zusätzlich jedem Befehl ein oder mehrere Bits hinzugefügt werden, die angeben, unter welchen Bedingungen der Befehl überhaupt auszuführen ist.

Zur weiteren Optimierung der Arbeitsgeschwindigkeit des Prozessors ist es dabei besonders bevorzugt, bei jedem der Befehle mit einem oder mehreren zusätzlichen Bits parallel zur Ausführung des Befehls die dem oder den Bits entsprechenden Flags (z.B. zero, carry, overflow) im Prozessor zu prüfen, wenn das entsprechende Bit gesetzt ist, und abhängig von diesem Ergebnis den Programmzähler des Prozessors entsprechend einzustellen.

Die Aufgabe der vorliegenden Erfindung wird ebenso durch eine Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur gelöst, in dem eine Vorrichtung zur Veränderung des Programmzählerstandes zur Ausführung der bedingten Sprünge vorgesehen ist.

Dabei ist es besonders bevorzugt, wenn die Vorrichtung zur Veränderung des Programmzählerstandes einen oder mehrere Eingänge für entsprechende zusätzliche Bits in den Maschinenbefehlen des Prozessors und einen oder mehrere Eingänge für die entsprechenden "Flag"-Signale aus dem Rechenwerk des Prozessors aufweist.

Es ist dabei besonders vorteilhaft, wenn sichergestellt ist, daß die entsprechenden zusätzlichen Bits aus den Maschinenbefehlen gleichzeitig mit den zugehörigen "Flag"-Signalen an der Vorrichtung zur Veränderung des Programmzählerstandes anliegen.

Vorzugsweise ist die Vorrichtung zur Veränderung des Programmzählerstandes mit einem Addierwerk ausgerüstet.

Die vorliegende Erfindung wird im folgenden anhand der in der Anlage beigefügten Zeichnungen näher erläutert. Es zeigen:

Fig. 1 den Arbeitsablauf eines Prozessors mit zweistufiger Pipeline;

Fig. 2 den Aufbau eines erfindungsgemäßen 22 Bit langen Maschinenbefehls;

Fig. 3 den Aufbau eines erfindungsgemäßen 25 Bit langen Maschinenbefehls;

Fig. 4 eine schematische Darstellung einer erfindungsgemäßen Vorrichtung zur Veränderung des Programmzählerstandes zur Ausführung bedingter Sprünge;

Fig. 5 eine weitere erfindungsgemäße Vorrichtung zur Veränderung des Programmzählerstandes zur Ausführung bedingter Sprünge;

Fig. 6 eine schematische Darstellung des Gesamtaufbaus eines Prozessors mit "Pipelined"-Architektur zur Ausführung bedingter Sprungbefehle mit der erfindungsgemäßen "branch-prediction"; und

Fig. 7 eine detaillierte Darstellung eines Prozessors mit Vorrichtungen zur erfindungsgemäßen "branch-prediction".

Die vorliegende Erfindung geht von einer "pipelined"-Architektur für einen Prozessor aus. Diese ist beispielsweise in dem Buch "Computer Organisation and Design" von Pattersen & Hennessy beschrieben.

Kurz gefaßt bedeutet die "Pipelined"-Architektur folgendes:

Normalerweise wird jeder Maschinenbefehl von einem Prozessor mittels folgender Operationen abgearbeitet:

1. Instruction fetch (Laden des Befehls)
2. Instruction decoding (Dekodieren des Befehls)
3. Execution (Ausführung des Befehls)
4. Write back (Zurückschreiben der Ergebnisse)

Es ist bereits im Stand der Technik bekannt, diese Operationen teilweise parallel ablaufen zu lassen, indem ein Befehl beispielsweise gerade ausgeführt wird, während bereits der nächste Befehl dekodiert wird. Diese Vorgehensweise ist in der Figur 1 für eine zweistufige Pipeline dargestellt.

Ein Prozessor nutzt die "Pipeline" also, um im Schnitt einen Befehl pro Prozessorzyklus zu verarbeiten.

Diese "Pipelined"-Architektur des Prozessors führt allerdings dann zu Problemen, wenn bedingte Sprungbefehle ausgeführt werden sollen. Dieses Problem wird in der Fachsprache "branch harzard" genannt. Dies bedeutet, daß ein "branch"-Befehl, also ein bedingter Sprungbefehl, erst nach Ausführung des vorigen Befehls zeigen kann, ob der nächste Befehl weiter bearbeitet oder auf eine andere Zieladresse gesprungen werden soll.

Im Stand der Technik löst man dieses Problem, indem der Takt nach dem bedingten Sprungbefehl mit einem "No operation"-Befehl, also einem Befehl, einen Prozessorzyklus zu warten, gefüllt wird. Dann ist zwar auf jeden Fall sichergestellt, daß das Programm richtig weiterläuft, man verliert aber einen Prozessorzyklus und damit die maximal mögliche Rechenleistung. Der bisherige Stand der Technik soll anhand der folgenden Beispiele, die jeweils die Berechnung des Absolut-Betrags einer Zahl behandeln, näher erläutert werden:

Zum einen gibt es die Möglichkeit der bedingten Ausführung, also beispielsweise:

```
/*  A = |B|  */
LOAD  R1   B
COMPARE R1  0  /*wenn B ≥ 0, carry = 0 */
NEGATIVE R1  on-carry  /* negieren wenn carry = 1 */
STORE  R1  A
```

Diese Art der Ausführung ist jedoch nur möglich, wenn nur ein einziger Befehl bedingt ausgeführt werden muß, und dieser Befehl keinen Sprung enthält. Bei komplexeren Funktionen oder Aufgaben, die nicht mehr nur mit einem Befehl dargestellt werden können, muß jeweils ein bedingter Sprung erfolgen, wie dies im folgenden Programm dargestellt ist. Wie aus dem eingerahmten Programmabschnitt erkennbar ist, muß nach den beiden Sprungbefehlen ein "no operation"-Befehl eingefügt werden (im Falle einer zweistufigen Pipeline, bei längeren Pipelines entsprechend mehr "no operation"-Befehle:

```
        LOAD        R1      B
        COMPARE     R1      0
        JUMP ON CARRY       L1
        JUMP        L2
        NO OP
L1:     NEGATIVE    R1
L2:     STORE       R1  A
```

Schließlich gibt es im Stand der Technik noch die Möglichkeit der sogenannten spekulativen Ausführung. Das bedeutet, daß man einfach eine Möglichkeit ausführt, und hofft, mit einer Wahrscheinlichkeit von etwas mehr als 50 % die richtige Fortsetzung zu treffen. Dies erfordert aber einen ganz erheblichen Hardware-Aufwand, da ja dann im Falle des Nichtzutreffens der Vermutung einige Befehle "rückabgewickelt" werden müssen. Außerdem gehen trotzdem Prozessorzyklen verloren, wenn man sich "verschätzt" hat.

Gemäß dem Stand der Technik gab es also bisher keine geeignete Lösung für dieses Problem, daß ein solcher "branch harzard", also ein Problem bei der bedingten Verzweigung, einen Verlust an Arbeitszyklen des Prozessors in einer "Pipelined"-Architektur bewirkte. Erfindungsgemäß wird nun durch eine Kombination eines Befehls mit "bedingter Ausführung" und

eines "Sprungarithmetik"-Befehls dieses Problem folgendermaßen gelöst:

Hier soll wieder ein einfaches Beispiel betrachtet werden, nämlich der Befehl "Addiere R2 zu R1, wenn R1 dann =0 ist, springe nach L1". Dieses Programm wird in "C" wie folgt geschrieben:

```
R1  = R1 + R2
if (R1 = = 0)
     GO TO L1
L1: ......
```

Erfindungsgemäß wird dafür der Maschinenbefehl ADD R1, R2, #JMP, ON ZERO, verwendet. #JMP bedeutet dabei die relative Sprungadresse zum Einsprungspunkt L1.

Damit erweitern wir einmal den Befehl um eine "Post-Condition" zur bekannten "Pre-condition". Zum Beispiel: P1, ADD R1, R2, #JMP, Q1.

Dabei bedeutet P1: Ausführen von R1=R1+R2 wenn P1 erfüllt ist. Erfindungsgemäß bedeutet Q1: Ausführen von Sprung um JMP wenn Q1 nach der Berechnung von R1=R1+R2 erfüllt ist.

Damit könnte man das folgende "C"-Programm:

```
if (A=1)
B = A;
else
C = A;
```

folgendermaßen in Maschinencode übersetzen:

```
LOAD          R1  A
Q1  TEST      R1  1 # L  /* if A=1 jump L */
P1  Q1  STORE R1  B
```

```
STORE          R1   C
```

Erfindungsgemäß können also in der Befehlscodierung sowohl Bits für "Pre-Conditions" als auch Bits für "Post-Conditions" vorgesehen werden, wie dies beispielsweise in Fig. 2 und 3 dargestellt ist.

Fig. 2 zeigt dabei ein vereinfachtes Beispiel mit einem lediglich 22 Bit langen Befehl, wobei ein Bit 1 für die "Pre-Condition" ein Bit 2 für die "Post-Conditions", 8 Bit 3 bis 10 für den relativen Sprungwert (Displacement) und dann wie üblich je drei Bits für die beiden Registeradressen und 6 Bit für den Befehlscode vorgesehen sind.

In der Realität ist es üblicherweise erforderlich, mehrere Bedingungen als "Pre-condition" und "Post-Condition" zu prüfen. Es müssen deshalb entsprechend mehr Bits vorgesehen werden, wie dies in Fig. 3 dargestellt ist.

In Fig. 3 enthalten die Bits 0 bis 1 die Informationen für Post condition, die Bits 2 und 3 Information für Pre-Conditions, die Bits 4 bis 10 die relative Sprungadresse, d.h. die Sprungweite.

Besonders wirkungsvoll läßt sich das erfindungsgemäße Verfahren im Zusammenhang mit einer Programmschleife einsetzten, beispielsweise für das folgende "C"-Programm:

```
für (i=1; i < 5; i++) {
     x [i] = i;
}      /*  C-Programm */
```

Erfindungsgemäß kann dies dann in das folgende erheblich vereinfachte Maschinenprogramm umgesetzt werden:

```
Load    R1   5
Load    R2   X   /*Adresse von X[5]*;
```

```
L1= STORE_INDEXED   R2   R1   /*  x[i] = i */
Q1  DECREMENT       R1   #1   L1
          ADD       R2    1     /*i = i+1*/
```

Dabei bedeutet die "Post condition Q1": Bedingter Sprung, wenn das Ergebnis R1=R1-1 nicht 0 ist.

Ein weiteres Beispiel für die erfindungsgemäß erzielbaren Vereinfachungen bei der Programmierung ist das im folgenden dargestellte Programm für die Abarbeitung eines Ringpuffers.

Gemäß dem Stand der Technik hätte dieses Programm wie folgt programmiert werden müssen:

```
TST (R3) #buffer_end    // ring buffer end reached
BNZ NEXT                // if no
NOP
LDI (R3) #buffer_start // else set the pointer to buffer
                          again
```

Erfindungsgemäß genügen statt dessen die folgenden beiden Befehle:

```
TST (R3)  #buffer_end
LDI (R3)  #buffer_start
```

Es ist jedoch zu beachten, daß diese erfindungsgemäße Lösung nicht für alle Schleifenstrukturen anwendbar ist. Schleifenstrukturen aller Art können jedoch erfindungsgemäß wie folgt programmiert werden:

```
LDI (R4) #loop_cnt_minus_1 // init loop counter
WHILE_LOOP:
     FIRST_PC                    // code sequency
     SUBI (R4)  #1 #loop_flag // decrement by 1 and in-
                                 dicate loop end
```

```
BNZ WHILE_LOOP          // if not zero go to loop
                           begin
```

Erfindungsgemäß wird anstelle des üblichen Subtraktions-Maschinenbefehls SUB ein Maschinenbefehl SUBI verwendet, der erweitert ist, so daß er ein Flag-Bit aufweist, welches dazu benutzt wird, einen Zyklus vor dem bedingten Sprungbefehl BNZ anzuzeigen, was die richtige Verzweigung beim bedingten Sprung ist, so daß im Falle einer zweistufigen Pipeline überhaupt kein Verlust an Prozessorzyklen auftritt. Der Befehl LDI zeigt einen Schleifenbeginn an.

Die typische Lösung zur Vermeidung des "branch harzard" beruht darauf, eine Vorhersage über das zu erwartende Sprungziel des bestimmten Sprunges zu machen.

Die Implementierung einer Schleife erfordert im allgemeinen diese drei Schritte:

1. Initialisiere den Schleifenzähler
2. Dekrementiere oder inkrementiere den Schleifenzähler
3. Springe am Ende der Schleife

Der Zyklenverlust bei dem bedingten Sprung beruht darauf, daß die nächste Instruktion, die nach dem Sprung ausgeführt wird abhängig von der Erfüllung der Schleifenbedingung ist. Diese Tatsache führt dazu, daß nach dem bedingten Sprungbefehl der Dummy-Befehl NOP eingefügt werden muß. Durch Verwendung eines Schleifen-Flags in einem Rechenbefehl wie ADD oder SUB kann die Schleifenbedingung am Ende der Ausführung des Additions- oder Subtraktionsbefehls geprüft werden. Dann kann das "Zero-flag", d.h. die Anzeige des Rechenwerks, daß es auf 0 steht, geprüft werden, um zu entscheiden, auf welche Adresse der Programmzähler des Prozessors gesetzt werden sollte. Das "LOOP-flag" kann als "ENABLE-DISABLE-flag" oder allgemeiner als Adressverschiebung interpretiert werden.

Figur 4 zeigt das einfachste Grundprinzip für die erfindungsgemäße Implementierung eines "LOOP"-flags.

Der Programmspeicher 10 wird hierbei über einen Multiplexer 12 mit dem Programmzähler 14 verbunden. Der Ausgang des Programmzählers (PC) 14 ist mit einem logischen Gatter 16 verbunden, welches den Ausgangswert des Programmzählers mit einer Konstante oder dem LOOP-flag verknüpft. Der Ausgang dieser Logik-Schaltung 16 ist mit dem einen Eingang des Multiplexers (MUX) 12 verbunden, dessen anderer Eingang ja mit dem Programmspeicher 10 verbunden ist, und dessen Ausgang mit dem Programmzähler 14 verbunden ist. Der Multiplexer 12 wird über ein Steuersignal (Control) vom Prozessor gesteuert.

Eine weitere Verbesserung der Erfindung erlaubt den Verzicht auf den Sprungbefehl, indem der Beginn der Schleife gepuffert wird:

```
        LDP  (R4) #loop_cnt_minus_1
WHILE_LOOP:
          FIRST_PC
SUBI (R4) #1 #Loop-flag
NEXT_INS:
```

Hierbei ist ein zuätzlicher Befehl LDP erforderlich, der anzeigt, daß eine Schleife beginnt. Die nächste Programmcodeadresse wird dann als Schleifenbeginn gepuffert. Das selbe Resultat könnte man auch erzielen, indem man den Befehl LDI verwendet und den nächsten Programmzählerwert explizit in den Puffer lädt. Hierdurch wird aber natürlich wieder ein zusätzlicher Befehl benötigt. Der Befehl SUBI weist ein Loop-flag auf, welches dazu dient, anzuzeigen, welches die richtige Verzweigung bei dem bedingten Sprung ist. Das zero-flag wird geprüft, um zu entscheiden, ob man zum Beginn der Schleife zurückspringen soll, oder die nächste Instruktion (NEXT_INS) ausführen soll, die durch #-Loop-Flag angezeigt ist.

Für diese vereinfachte Bearbeitung von Schleifenstrukturen ist eine etwas kompliziertere Struktur der erfindungsgemäßen Schaltung erforderlich, wie sie in Fig. 5 dargestellt ist.

Ähnlich wie in Fig. 4 ist auch hier ein Programmspeicher 10 vorgesehen, der mit dem Eingang eines Multiplexers 12 verbunden ist, dessen Ausgang wiederum mit dem Programmzähler (PC) 14 verbunden ist. Der Ausgang des Programmzählers (PC) 14, ist ebenfalls mit einem logischen Gatter 16 verbunden, welches den Ausgangswert des Programmzählers mit dem Loop-Flag verknüpft. Der Ausgang dieser Logik-Schaltung 16 ist mit einem weiteren Eingang des Multiplexers (MUX) 12 verbunden. Im vorliegenden Fall weist jedoch der Mulitplexer 12 einen weiteren Eingang auf, der mit einem Puffer 18 verbunden ist, dessen Eingang mit dem Wert des Programmzählers 14 geladen werden kann. Auf diese Weise erübrigt sich der explizite Befehl "Lade den nächsten Programmzählerstand in den Puffer".

Die Fig. 6 zeigt den gesamten Aufbau eines Prozessors mit der Fähigkeit, die erfindungsgemäßen Befehle abzuarbeiten. Gleiche Elemente wie in den Fig. 4 und 5 sind auch hier mit gleichen Bezugszeichen versehen. Der Programmzähler (PC) 14 greift wiederum auf den Programmcodespeicher 10, und dabei jeweils auf die abzuarbeitende Programmzeile zu. Vom Programmspeicher 10 wird der entsprechende Instruction Code (Befehlscode) dem Befehlsdecoder (IDEC) 20 zugeführt. Dieser gibt die entsprechenden Steuerbefehle an das Rechenwerk (ALU) 22 und an den Registersatz 24 weiter. Die Inhalte der Register werden dann nach Bedarf in das Rechenwerk 22 geladen, oder von dort wieder zurückgeschrieben, wie dies mit den Pfeilen angedeutet ist. Die Flag-Signale zero, carry und overflow des Rechenwerks 22 werden gleichzeitig sowohl dem Befehlsdecoder (IDEC) 20 als auch dem Steuereingang des Multiplexers (MUX) 12 zugeführt. Die beiden Eingänge des Multiplexers 12 sind mit dem Wert 1 und mit dem vom Befehlsdecoder 20 gelieferten relativen Sprungwert #JMP belegt. Der Ausgang des Multiplexers 12 ist mit einem Addierwerk 16 verbunden,

dessen anderer Eingang mit dem Ausgang des Programmzählers 14 verbunden ist.

Bei mehr als zwei Pipelined-Stufen ist zu beachten, daß die Flag-Signale zero, carry, overflow und der zugehörige relative Sprungwert #JMP gleichzeitig am Multiplexer 12 anliegen müssen. Bei einer zweistufigen Pipeline, wie bei dem vorliegenden Ausführungsbeispiel beschrieben, ist dies jedoch nicht erforderlich. Im folgenden wird nun die entsprechende Befehlscodierung mit der erfindungsgemäßen "Post-condition" beschrieben. Hierzu wenden wir uns nochmals der Fig. 2 zu, die den einfachstmöglichen erfindungsgemäßen Befehlssatz mit einer Länge von 22 Bit darstellt.

Die obersten 6 Bit (21 bis 16) enthalten dabei den Befehlscode (OPCODE), beispielsweise: Addition. Die nächsten drei Bits enthalten die Adresse des ersten Registers (REG A)mit drei Bit Länge (übliche Prozessoren verwenden meist nicht mehr als 8 Register) auf den Bits 15, 14, 13, sodann folgt die Registeradresse des zweiten, im vorliegenden Fall zu addierenden Registers (REG B) auf den Bits 12, 11 und 10.

Das Rechenwerk des Prozessors wird bei diesem Befehl also den Inhalt der Register A und B addieren und ins Register ablegen. Erfindungsgemäß sind diesem Befehl nun weitere Bits angefügt, nämlich die Bits 9 bis 2 (displacement), die die relative Sprungweite bei einem folgenden bedingten Sprung angeben. Sodann folgen die Condition-Bits 1 und 0, wobei das Bit 1 (Post) die Post-condition angibt, während das Bit 0 (PRE) die Pre-condition angibt.

Der Bearbeitungsablauf ist dabei nun folgendermaßen: Der Befehl muß abgeholt und decodiert werden. Dazu startet der Prozessor an einem bestimmten Programmzählerstand, z.B. PC=0.

Mit diesem Programmzählerstand wird ein Befehl von 22 Bit aus dem Programmspeicher abgeholt, der an der diesem Programmzählerstand entsprechenden Adresse im Speicher steht.

Der Befehl wird sodann vom Instruction-Decoder (IDEC) 20 verarbeitet.

Dabei wird zuerst geprüft, ob das entsprechende Pre-condition-Bit gesetzt ist. Wenn dies der Fall ist, wird der Befehl beim Nichterfüllen der entsprechenden Pre-condition gar nicht ausgeführt.

Der Unterschied der vorliegenden Erfindung zum Stand der Technik liegt in den Post-condition-Bits.

Aus diesem Post-condition-Bits wird das Signal "BR_CTR" erzeugt. Gleichzeitig wird die Addition folgendermaßen durchgeführt:

Ein Steuersignal ALU-CTR sowie die Lese- und Schreibadressen und Enable-Signale für das Rechenwerk werden erzeugt. Zugleich stellt der Instruction-Decoder 20 die relative Sprungweite "BR" zur Verfügung. Das "BR-CTR"-Signal steuert die Verzweigungskontrolle nach folgenden Vorgaben an:

1. Kein Sprung, wenn Post-condition-Bit=0, also PCNEW=PCOLD+1

2. Wenn Post-condition-Bit=1 und die Bedingung erfüllt wird, z.B. zero-flag=1, dann wird ein relativer Sprung ausgeführt. Der Programmzähler 14 wird also auf den neuen Wert PCNEW=PCOLD+BR gesetzt.

Wenn das Post-condition-Bit zwar =1 ist, die Bedingung aber nicht erfüllt wird, wird ebenfalls kein Sprung durchgeführt, also: PCNEW=PCOLD+1.

Es ist möglich, mehr als ein Post-condition-Bit zu verwenden, wie dies beispielsweise in Fig. 3 dargestellt ist. Es können dann mehr Bedingungen geprüft werden (beispielsweise zero, carry, overflow).

Erfindungsgemäß wird also erstmals gleichzeitig Steuerinformation für das Rechenwerk und Information zu Sprungzieladressen gleichzeitig vom Instruction-Decoder 20 beim Decodieren der Befehle bereitgestellt.

Nunmehr wird der Befehl ausgeführt und ggf. verzweigt.

Dazu wird die Aktion vom Rechenwerk (ALU) ausgeführt. Das Ergebnis wird in das entsprechende Register zurückgeschrieben. Gleichzeitig liegen die entsprechenden zero-, carry- usw. -flags am Ausgang des Rechenwerks an.

Der Verzweigungssteuerung werden dabei die Bits für die einzelnen Flags, "BRCTR" und der Wert "BR" zum gleichen Takt zur Verfügung gestellt. Wie in Fig. 7 dargestellt, erzeugt dann die Steuereinheit "Cond" 26 zwei Steuersignale S1 und S2. S1 steuert an, entweder keinen Sprung vorzunehmen, oder einen relativen Sprung auszurechnen. S2 schaltet dann die relative Sprungadresse "PCNEW" durch den Multiplexer 12 durch.

Im Ergebnis spart man einen zusätzlichen Befehl für den Sprung zusätzlich zu dem entsprechenden Arithmetikbefehl. Dadurch kann man eine Verringerung der Anzahl der erforderlichen Befehle erreichen und erhöht damit den Durchsatz des Prozessors.

Der Aufbau eines Prozessors zur Bearbeitung von Befehlen mit den erfindungsgemäßen "Post-condition-Bits" ist in Fig. 7 im einzelnen dargestellt. Gleiche Ziffern wie in den Fig. 4, 5 und 6 verweisen auf gleiche Einheiten.

Auch in Fig.7 ist ein Programmzähler 14 vorgesehen, der einen Befehlsspeicher (CODEROM) 10 adressiert. Von dort werden die Befehle mit einer Befehlsbreite von 22 Bit dem Befehlsdecodierer (IDEC) 20 zugeführt. Dieser erzeugt die üblichen Signale zur Ansteuerung der Register 24 und des Rechenwerks (ALU) 22. Erfindungsgemäß erzeugt er jedoch auch zusätzlich die Signale "BR" (dieses Signal umfaßt mehrere Bits) und gibt die relative Sprungweite an, sowie das Signal "BR-CTR", welches angibt, daß ein bedingter Sprung abzuarbeiten ist, und die entsprechenden Flag-Bits des Rechenwerks abzuprüfen sind.

Das Rechenwerk 22 liefert an seinem Ausgang Ergebnisse (result) und die entsprechenden Flags, die bestimmten Bedingungen (z.B. 0=zero, Überlauf=overflow, Übertrag=carry usw.) darstellen. Die Ergebnisse (result) können natürlich auch den Registern 24 wieder zugeführt werden. Die "BR_CTR"-Signale und die Flags aus der ALU werden einer weiteren Logikeinheit (Cond) 26 zugeführt. Diese erzeugt in Abhängigkeit von den entsprechenden BR_CTR-Signalen und den zugehörigen Flags Signale S1 und S2, die den Multiplexer 12 und einen Schalter vor dem einen Eingang des Addierwerks 16 steuern. Dieser Schalter schaltet abhängig von der Erfüllung der Flagbedingungen zwischen 1 und "BR" um. Der andere Eingang dieses Addierwerks ist mit dem Ausgang des Programmzählers 14 verbunden.

Auf diese erfindungsgemäße Weise kann mit relativ wenig technischem Zusatzaufwand am Prozessor eine wesentlich schnellere Abarbeitung bedingter Sprünge durchgeführt werden.

Patentansprüche

1. Verfahren zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur, d a d u r c h g e k e n n z e i c h n e t, daß jedem Befehl, nach dem ein bedingter Sprung ausgeführt werden soll, ein oder mehrere zusätzliche Bits hinzugefügt werden, die angeben, unter welcher Bedingung der bedingte Sprung auszuführen ist.

2. Verfahren nach Anspruch 1, d a d u r c h g e k e n n z e i c h n e t, daß zusätzlich zu jedem Befehl, nach dem ein bedingter Sprung ausgeführt werden soll, die entsprechende Sprungadresse zugefügt wird.

3. Verfahren nach Anspruch 1 oder Anspruch 2, d a d u r c h g e k e n n z e i c h n e t, daß zusätzlich jedem Befehl ein oder mehrere Bits hinzugefügt werden, die angeben, unter welchen Bedingungen der Befehl überhaupt auszuführen ist.

4. Verfahren nach einem der Ansprüche 1 bis 3, d a d u r c h g e k e n n z e i c h n e t, daß bei jedem der Befehle mit einem oder mehreren zusätzlichen Bits, parallel zur Ausführung des Befehls die dem oder den Bits entsprechenden Flags (z.B. zero, carry, overflow) im Prozessor geprüft werden, wenn das entsprechende Bit gesetzt ist, und abhängig von diesem Ergebnis der Programmzähler (PC) des Prozessors entsprechend eingestellt wird.

5. Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur, d a d u r c h g e k e n n z e i c h n e t, daß eine Vorrichtung zur Veränderung des Programmzählerstandes zur Ausführung der bedingten Sprünge vorgesehen ist.

6. Vorrichtung nach Anspruch 5, d a d u r c h g e k e n n z e i c h n e t, daß die Vorrichtung zur Veränderung des Programmzähelerstandes einen oder mehrere Eingänge für entspre-

chende zusätzliche Bits in den Maschinenbefehlen des Prozessors und einen oder mehrere Eingänge für die entsprechenden "flag"-Signale aus dem Rechenwerk des Prozessors aufweist.

7. Vorrichtung nach Anspruch 6, dadurch gekennzeichnet, daß die entsprechenden zusätzlichen Bits aus den Maschinenbefehlen gleichzeitig mit den zugehörigen "flag"-Signalen an der Vorrichtung zur Veränderung des Programmzählerstandes anliegen.

8. Vorrichtung nach einem der Ansprüche 5 bis 7, dadurch gekennzeichnet, daß die Vorrichtung zur Veränderung des Programmzählerstandes ein Addierwerk umfaßt.

Zusammenfassung

Verfahren und Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur

Verfahren und Vorrichtung zur Bearbeitung bedingter Sprungbefehle in einem Prozessor mit "Pipelined"-Architektur, wobei jedem Befehl, nach dem ein bedingter Sprung ausgeführt werden soll, ein oder mehrere zusätzliche Bits hinzugefügt werden, die angeben, unter welcher Bedingung der bedingte Sprung auszuführen ist. Zusätzlich kann die Vorrichtung eine Vorrichtung zur Veränderung des Programmzählerstandes in Abhängigkeit von den zusätzlichen Bits zur Ausführung der bedingten Sprünge umfassen.

Figur 7

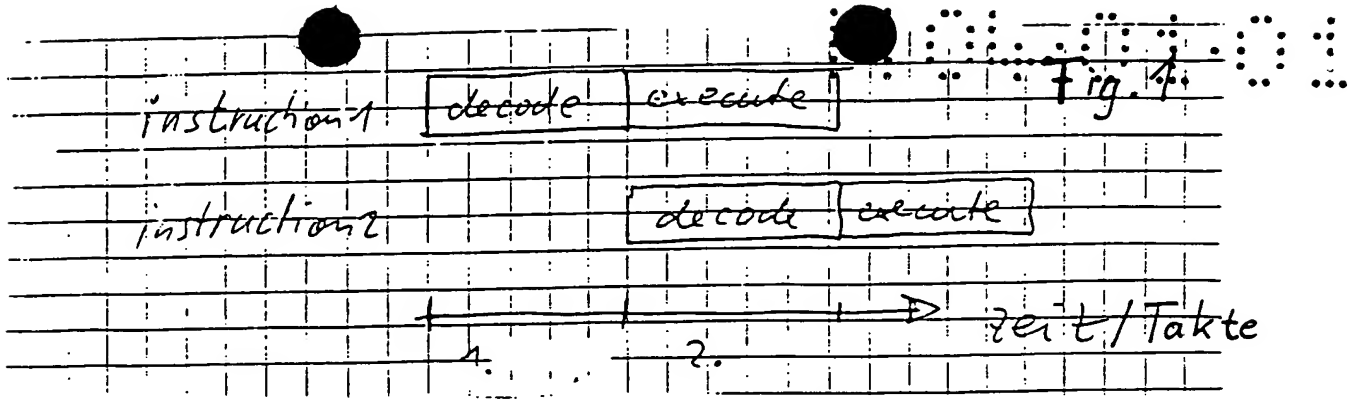


Fig. 1

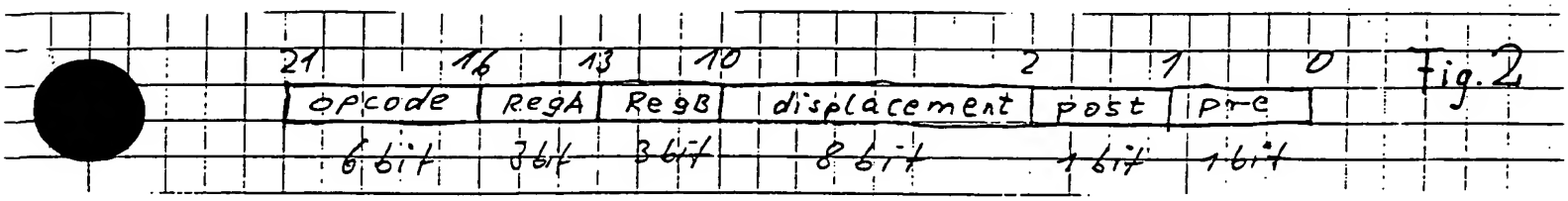


Fig. 2

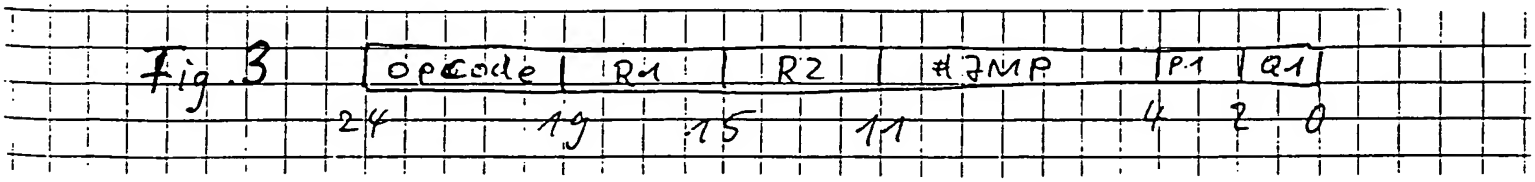


Fig. 3

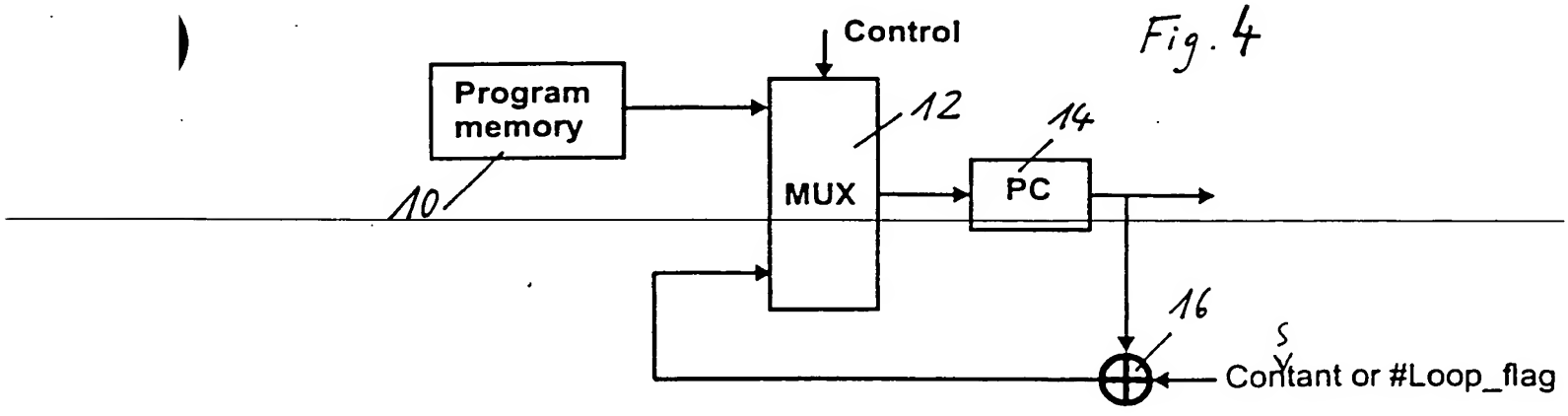


Fig. 4

Fig. 5.01

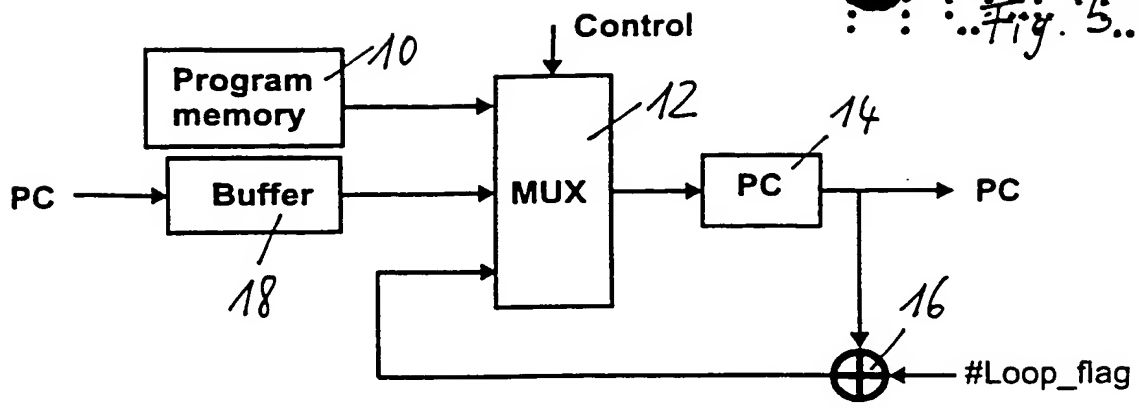

Control
Program memory
10
12
14
PC
Buffer
MUX
PC
PC
18
16
#Loop_flag

Fig. 6

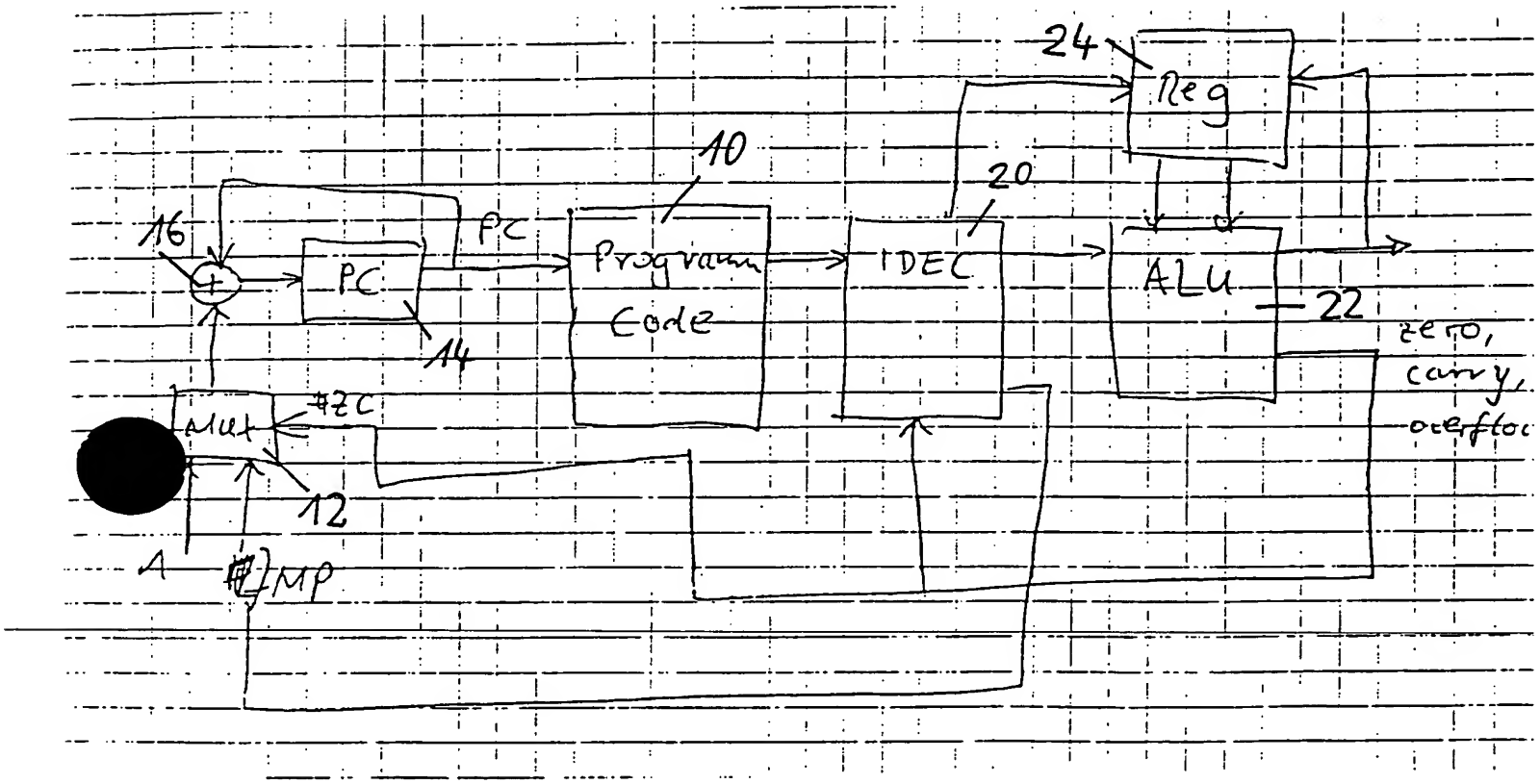
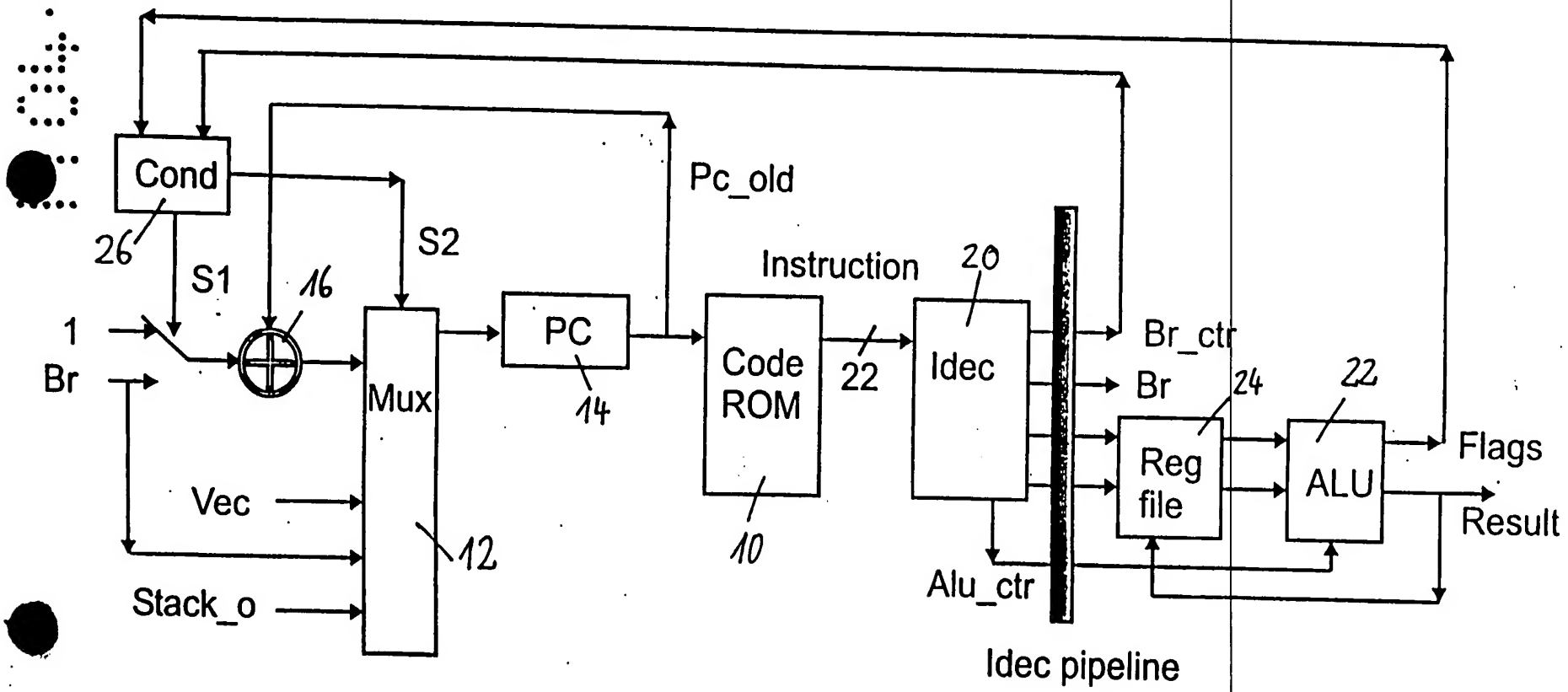

24
Reg
10
20
16
PC
PC
Program Code
IDEC
ALU
22
14
zero,
carry,
overflo
12
A
#JMP

Fig. 7


Cond
26
S1
S2
16
1
Br
Mux
PC
14
Pc_old
Code ROM
10
Instruction
22
20
Idec
Br_ctr
Br
24
Reg file
22
ALU
Flags
Result
Vec
12
Stack_o
Alu_ctr
Idec pipeline